- A doubly linked list permits backward traversal and deletion from the end of the list.
- An iterator is a reference, encapsulated in a class object, that points to a link in an associated list.
- Iterator methods allow the user to move the iterator along the list and access the link currently pointed to.
- An iterator can be used to traverse through a list, performing some operation on selected links (or all links).

Triangular Numbers

It's said that the Pythagorians, a band of mathematicians in ancient Greece who worked under Pythagoras (of Pythagorian theorem fame), felt a mystical connection with the series of numbers 1, 3, 6, 10, 15, 21, … (where the … means the series continues indefinitely). Can you find the next member of this series?

The `n`th term in the series is obtained by adding `n` to the previous term. Thus the second term is found by adding 2 to the first term (which is 1), giving 3. The third term is 3 added to the second term (which is 3), giving 6, and so on. The numbers in this series are called *triangular numbers* because they can be visualized as a triangular arrangements of objects, shown as little squares in Figure 6.1.

Finding the nth Term Using a Loop

Suppose you wanted to find the value of some arbitrary nth term in the series; say the fourth term (whose value is 10). How would you calculate it? Looking at Figure 6.2, you might decide that the value of any term can be obtained by adding up all the vertical columns of squares.

In the fourth term, the first column has four little squares, the second column has three, and so on. Adding 4+3+2+1 gives 10.

Figure 6.1: The triangular numbers

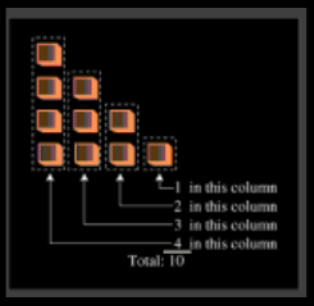

Figure 6.2: Triangular number as columns

The following `triangle()` method uses this column-based technique to find a triangular number. It sums all the columns, from a height of `n` to a height of 1.

```
int triangle(int n)
   {
   int total = 0;

   while(n > 0)           // until n is 1
      {
      total = total + n;  // add n (column height) to total
      --n;                // decrement column height
      }
   return total;

   }
```

The method cycles around the loop `n` times, adding `n` to `total` the first time, `n-1` the second time, and so on down to 1, quitting the loop when `n` becomes 0.

Finding the nth Term Using Recursion

The loop approach may seem straightforward, but there's another way to look at this problem. The value of the nth term can be thought of as the sum of only two things, instead of a whole series. These are

1. The first (tallest) column, which has the value n.

2. The sum of all the remaining columns.

This is shown in Figure 6.3.

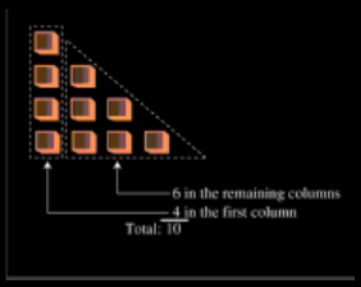

Figure 6.3: Triangular number as column plus triangle

Finding the Remaining Columns

If we knew about a method that found the sum of all the remaining columns, then we could write our `triangle()` method, which returns the value of the `n`th triangular number, like this:

```
int triangle(int n)
   {
   return( n + sumRemainingColumns(n) );  // (incomplete
version)

   }
```

But what have we gained here? It looks like it's just as hard to write the `sumRemainingColumns()` method as to write the `triangle()` method in the first place.

Notice in Figure 6.3, however, that the sum of all the remaining columns for term `n` is the same as the sum of *all* the columns for term `n-1`. Thus, if we knew about a method that summed all the columns for term `n`, we could call it with an argument of `n-1` to find the sum of all the remaining columns for term `n`:

```
int triangle(int n)
   {
   return( n + sumAllColumns(n-1) );  // (incomplete version)

   }
```

But when you think about it, the `sumAllColumns()` method is doing exactly the same thing the `triangle()` method is doing: summing all the columns for some number `n` passed as an argument. So why not use the `triangle()` method itself, instead of some other method? That would look like this:

```
int triangle(int n)
   {
   return( n + triangle(n-1) );  // (incomplete version)

   }
```

It may seem amazing that a method can call itself, but why shouldn't it be able to? A method call is (among other things) a transfer of control to the start of the method. This transfer of control can take place from within the method as well as from outside.

Passing the Buck

All this may seem like passing the buck. Someone tells me to find the 9th triangular number. I know this is 9 plus the 8th triangular number, so I call Harry and ask him to find the 8th triangular number. When I hear back from him, I'll add 9 to whatever he tells me, and that will be the answer.

Harry knows the 8th triangular number is 8 plus the 7th triangular number, so he calls Sally and asks her to find the 7th triangular number. This process continues with each person passing the buck to another one.

Where does this buck-passing end? Someone at some point must be able to figure out an answer that doesn't involve asking another person to help them. If this didn't happen, there would be an infinite chain of people asking other people questions; a sort of arithmetic Ponzi scheme that would never end. In the case of `triangle()`, this would mean the method calling itself over and over in an infinite series that would paralyze the program.

The Buck Stops Here

To prevent an infinite regress, the person who is asked to find the first triangular number of the series, when n is 1, must know, without asking anyone else, that the answer is 1. There are no smaller numbers to ask anyone about, there's nothing left to add to anything else, so the buck stops there. We can express this by adding a condition to the `triangle()` method:

```
int triangle(int n)
   {
   if(n==1)
      return 1;
   else
      return( n + triangle(n-1) );

   }
```

The condition that leads to a recursive method returning without making another recursive call is referred to as the *base case*. It's critical that every recursive method have a base case to prevent infinite recursion and the consequent demise of the program.

The `triangle.java` Program

Does recursion actually work? If you run the `triangle.java` program, you'll see that it does. Enter a value for the term number, `n`, and the program will display the value of the corresponding triangular number. Listing 6.1 shows the `triangle.java` program.

Listing 6.1 The triangle.java Program

```
// triangle.java
// evaluates triangular numbers
// to run this program: C>java TriangleApp
import java.io.*;                 // for I/O
////////////////////////////////////////////////////////////////
class TriangleApp
   {
   static int theNumber;

   public static void main(String[] args) throws IOException
      {
      System.out.print("Enter a number: ");
      System.out.flush();
      theNumber = getInt();
      int theAnswer = triangle(theNumber);
      System.out.println("Triangle="+theAnswer);
      }  // end main()

//-------------------------------------------------------------
```

```
   public static int triangle(int n)
      {
      if(n==1)
         return 1;
      else
         return( n + triangle(n-1) );
      }

//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }

//--------------------------------------------------------------
-
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }

//--------------------------------------------------------------
-

   }  // end class TriangleApp
```

The `main()` routine prompts the user for a value for `n`, calls `triangle()`, and displays the return value. The `triangle()` method calls itself repeatedly to do all the work.

Here's some sample output:

```
Enter a number: 1000
Triangle = 500500
```

Incidentally, if you're skeptical of the results returned from `triangle()`, you can check them by using the following formula:

```
nth triangular number = (n e2+n)/2
```

What's Really Happening?

Let's modify the `triangle()` method to provide an insight into what's happening when it executes. We'll insert some output statements to keep track of the arguments and return values:

```
public static int triangle(int n)
   {
   System.out.println("Entering: n=" + n);
   if(n==1)
      {
```

```
        System.out.println("Returning 1");
        return 1;
        }
     else
        {
        int temp = n + triangle(n-1);
        System.out.println("Returning " + temp);
        return temp;
        }

     }
```

Here's the interaction when this method is substituted for the earlier `triangle()` method and the user enters 5:

```
Enter a number: 5


Entering: n=5
Entering: n=4
Entering: n=3
Entering: n=2
Entering: n=1
Returning 1
Returning 3
Returning 6
Returning 10
Returning 15

Triangle = 15
```

Each time the `triangle()` method calls itself, its argument, which starts at 5, is reduced by 1. The method plunges down into itself again and again until its argument is reduced to 1. Then it returns. This triggers an entire series of returns. The method rises back up, phoenixlike, out of the discarded versions of itself. Each time it returns, it adds the value of `n` it was called with to the return value from the method it called.

The return values recapitulate the series of triangular numbers, until the answer is returned to `main()`. Figure 6.4 shows how each invocation of the `triangle()` method can be imagined as being "inside" the previous one.

Notice that, just before the innermost version returns a 1, there are actually five different incarnations of `triangle()` in existence at the same time. The outer one was passed the argument `5`; the inner one was passed the argument `1`.

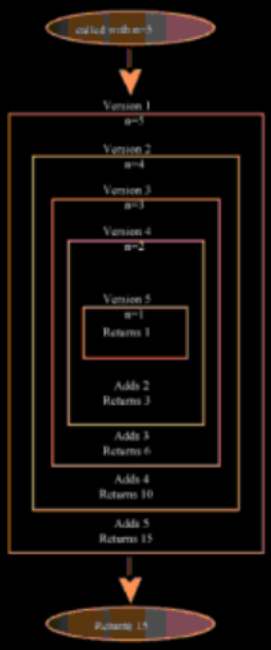

Figure 6.4: The recursive `triangle()` method

Characteristics of Recursive Methods

Although it's short, the `triangle()` method possesses the key features common to all recursive routines:

- It calls itself.
- When it calls itself, it does so to solve a smaller problem.
- There's some version of the problem that is simple enough that the routine can solve it, and return, without calling itself.

In each successive call of a recursive method to itself, the argument becomes smaller (or perhaps a range described by multiple arguments becomes smaller), reflecting the fact that the problem has become "smaller" or easier. When the argument or range reaches a certain minimum size, a condition is triggered and the method returns without calling itself.

Is Recursion Efficient?

Calling a method involves certain overhead. Control must be transferred from the location of the call to the beginning of the method. In addition, the arguments to the method, and the address to which the method should return, must be pushed onto an internal stack so that the method can access the argument values and know where to return.

In the case of the `triangle()` method, it's probable that, as a result of this overhead, the `while` loop approach executes more quickly than the recursive approach. The penalty may not be significant, but if there are a large number of method calls as a result of a recursive method, it might be desirable to eliminate the recursion. We'll talk about this more at the end of this chapter.

Another inefficiency is that memory is used to store all the intermediate arguments and

return values on the system's internal stack. This may cause problems if there is a large amount of data, leading to stack overflow.

Recursion is usually used because it simplifies a problem conceptually, not because it's inherently more efficient.

Mathematical Induction

Recursion is the programming equivalent of mathematical induction. Mathematical induction is a way of defining something in terms of itself. (The term is also used to describe a related approach to proving theorems.) Using induction, we could define the triangular numbers mathematically by saying

```
if n = 1

tri(n) = n + tri(n-1)     if n > 1
```

Defining something in terms of itself may seem circular, but in fact it's perfectly valid (provided there's a base case).

Factorials

Factorials are similar in concept to triangular numbers, except that multiplication is used instead of addition. The triangular number corresponding to n is found by adding n to the triangular number of n–1, while the factorial of n is found by multiplying n by the factorial of n–1. That is, the fifth triangular number is 5+4+3+2+1, while the factorial of 5 is 5*4*3*2*1, which equals 120. Table 6.1 shows the factorials of the first 10 numbers.

Table 6.1: Factorials

Number	Calculation	Factorial
0	by definition	1
1	1 * 1	1
2	2 * 1	2
3	3 * 2	6
4	4 * 6	24
5	5 * 24	120
6	6 * 120	720
7	7 * 720	5,040
8	8 * 5,040	40,320

9	9 * 40,320	362,880

The factorial of 0 is defined to be 1. Factorial numbers grow large very rapidly, as you can see.

A recursive method similar to `triangle()` can be used to calculate factorials. It looks like this:

```
int factorial(int n)
   {
   if(n==0)
      return 1;
   else
      return (n * factorial(n-1) );

   }
```

There are only two differences between `factorial()` and `triangle()`. First, `factorial()` uses an * instead of a + in the expression

```
n * factorial(n-1)
```

Second, the base condition occurs when `n` is 0, not 1. Here's some sample interaction when this method is used in a program similar to `triangle.java`:

```
Enter a number: 6

Factorial =720
```

Figure 6.5 shows how the various incarnations of `factorial()` call themselves when initially entered with `n=4`.

Calculating factorials is the classic demonstration of recursion, although factorials aren't as easy to visualize as triangular numbers.

Various other numerological entities lend themselves to calculation using recursion in a similar way, such as finding the greatest common denominator of two numbers (which is used to reduce a fraction to lowest terms), raising a number to a power, and so on. Again, while these calculations are interesting for demonstrating recursion, they probably wouldn't be used in practice because a loop-based approach is more efficient.

Figure 6.5: The recursive `factorial()` method

Anagrams

Here's a different kind of situation in which recursion provides a neat solution to a problem. Suppose you want to list all the anagrams of a specified word; that is, all possible letter combinations (whether they make a real English word or not) that can be made from the letters of the original word. We'll call this *anagramming* a word. Anagramming `cat`, for example, would produce

- `cat`
- `cta`
- `atc`
- `act`
- `tca`
- `tac`

Try anagramming some words yourself. You'll find that the number of possibilities is the factorial of the number of letters. For 3 letters there are 6 possible words, for 4 letters there are 24 words, for 5 letters 120 words, and so on. (This assumes that all letters are distinct; if there are multiple instances of the same letter, there will be fewer possible words.)

How would you write a program to anagram a word? Here's one approach. Assume the word has n letters.

1. Anagram the rightmost n–1 letters.
2. Rotate all n letters.
3. Repeat these steps n times.

To *rotate* the word means to shift all the letters one position left, except for the leftmost letter, which "rotates" back to the right, as shown in Figure 6.6.

Rotating the word n times gives each letter a chance to begin the word. While the selected letter occupies this first position, all the other letters are then anagrammed (arranged in every possible position). For cat, which has only 3 letters, rotating the remaining 2 letters simply switches them. The sequence is shown in Table 6.2.

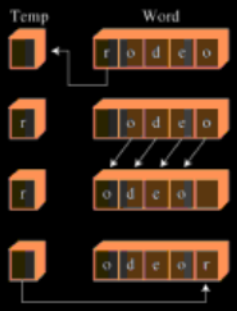

Figure 6.6: Rotating a word

Table 6.2: Anagramming the word cat

Word	Display Word?	First Letter	Remaining Letters	Action
cat	Yes	c	at	Rotate at
cta	Yes	c	Ta	Rotate ta
cat	No	c	at	Rotate cat
atc	Yes	a	Tc	Rotate tc
act	Yes	a	ct	Rotate ct
atc	No	a	Tc	Rotate atc
tca	Yes	t	ca	Rotate ca
tac	Yes	t	ac	Rotate ac
tca	No	t	ca	Rotate tca
cat	No	c	at	Done

Notice that we must rotate back to the starting point with two letters before performing a 3-letter rotation. This leads to sequences like cat, cta, cat. The redundant combinations aren't displayed.

How do we anagram the rightmost n–1 letters? By calling ourselves. The recursive `doAnagram()` method takes the size of the word to be anagrammed as its only parameter. This word is understood to be the rightmost n letters of the complete word. Each time `doAnagram()` calls itself, it does so with a word one letter smaller than before, as shown in Figure 6.7.

The base case occurs when the size of the word to be anagrammed is only one letter. There's no way to rearrange one letter, so the method returns immediately. Otherwise, it anagrams all but the first letter of the word it was given and then rotates the entire word. These two actions are performed n times, where n is the size of the word. Here's the recursive routine `doAnagram()`:

```
public static void doAnagram(int newSize)
   {
   if(newSize == 1)                     // if too small,
      return;                           // go no further
   for(int j=0; j<newSize; j++)         // for each position,
      {
      doAnagram(newSize-1);             // anagram remaining
      if(newSize==2)                    // if innermost,
         displayWord();                 // display it
      rotate(newSize);                  // rotate word
      }

   }
```

Each time the `doAnagram()` method calls itself, the size of the word is one letter smaller, and the starting position is one cell further to the right, as shown in Figure 6.8.

Figure 6.7: The recursive `doAnagram()` method

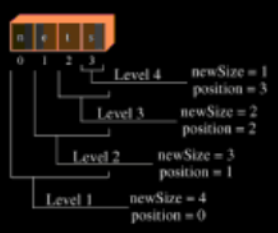

Figure 6.8: Smaller and smaller words

Listing 6.2 shows the complete `anagram.java` program. The `main()` routine gets a word from the user, inserts it into a character array so it can be dealt with conveniently, and then calls `doAnagram()`.

Listing 6.2 The anagram.java Program

```
// anagram.java
// creates anagrams
// to run this program: C>java AnagramApp
import java.io.*;                  // for I/O
////////////////////////////////////////////////////////////////
class AnagramApp
   {
   static int size;
   static int count;
   static char[] arrChar = new char[100];

   public static void main(String[] args) throws IOException
      {
      System.out.print("Enter a word: ");    // get word
      System.out.flush();
      String input = getString();
      size = input.length();                 // find its size
      count = 0;
      for(int j=0; j<size; j++)              // put it in array
         arrChar[j] = input.charAt(j);
      doAnagram(size);                       // anagram it
      }  // end main()

   //-------------------------------------------------------------
-
   public static void doAnagram(int newSize)
      {
      if(newSize == 1)                     // if too small,
         return;                           // go no further
      for(int j=0; j<newSize; j++)         // for each
position,
         {
         doAnagram(newSize-1);             // anagram remaining
         if(newSize==2)                    // if innermost,
            displayWord();                 // display it
         rotate(newSize);                  // rotate word
         }
      }

   //-------------------------------------------------------------
-
   // rotate left all chars from position to end
   public static void rotate(int newSize)
      {
```

```
      int j;
      int position = size - newSize;
      char temp = arrChar[position];       // save first letter
      for(j=position+1; j<size; j++)       // shift others left
         arrChar[j-1] = arrChar[j];
      arrChar[j-1] = temp;                 // put first on
right
      }

   //--------------------------------------------------------
-
   public static void displayWord()
      {
      if(count < 99)
         System.out.print(" ");
      if(count < 9)
         System.out.print(" ");
      System.out.print(++count + " ");
      for(int j=0; j<size; j++)
         System.out.print( arrChar[j] );
      System.out.print("   ");
      System.out.flush();
      if(count%6 == 0)
         System.out.println("");
      }

   //--------------------------------------------------------
-
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }

   //--------------------------------------------------------
---

   }  // end class AnagramApp
```

The `rotate()` method rotates the word one position left as described earlier. The `displayWord()` method displays the entire word and adds a count to make it easy to see how many words have been displayed. Here's some sample interaction with the program:

```
Enter a word: cats
1 cats     2 cast     3 ctsa     4 ctas     5 csat     6 csta
7 atsc     8 atcs     9 asct    10 astc    11 acts    12 acst
13 tsca    14 tsac    15 tcas    16 tcsa    17 tasc    18 tacs
19 scat    20 scta    21 satc    22 sact    23 stca    24 stac
```

(Is it only coincidence that scat is an anagram of cats?) You can use the program to anagram 5-letter or even 6-letter words. However, because the factorial of 6 is 720, this may generate more words than you want to know about.

Anagrams

Here's a different kind of situation in which recursion provides a neat solution to a problem. Suppose you want to list all the anagrams of a specified word; that is, all possible letter combinations (whether they make a real English word or not) that can be made from the letters of the original word. We'll call this *anagramming* a word. Anagramming `cat`, for example, would produce

- `cat`
- `cta`
- `atc`
- `act`
- `tca`
- `tac`

Try anagramming some words yourself. You'll find that the number of possibilities is the factorial of the number of letters. For 3 letters there are 6 possible words, for 4 letters there are 24 words, for 5 letters 120 words, and so on. (This assumes that all letters are distinct; if there are multiple instances of the same letter, there will be fewer possible words.)

How would you write a program to anagram a word? Here's one approach. Assume the word has n letters.

1. Anagram the rightmost n–1 letters.
2. Rotate all n letters.
3. Repeat these steps n times.

To *rotate* the word means to shift all the letters one position left, except for the leftmost letter, which "rotates" back to the right, as shown in Figure 6.6.

Rotating the word n times gives each letter a chance to begin the word. While the selected letter occupies this first position, all the other letters are then anagrammed (arranged in every possible position). For cat, which has only 3 letters, rotating the remaining 2 letters simply switches them. The sequence is shown in Table 6.2.

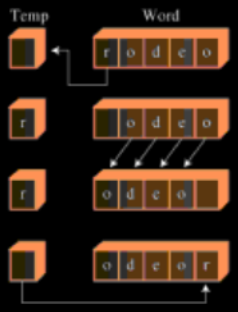

Figure 6.6: Rotating a word

Table 6.2: Anagramming the word cat

Word	Display Word?	First Letter	Remaining Letters	Action
cat	Yes	c	at	Rotate at
cta	Yes	c	Ta	Rotate ta
cat	No	c	at	Rotate cat
atc	Yes	a	Tc	Rotate tc
act	Yes	a	ct	Rotate ct
atc	No	a	Tc	Rotate atc
tca	Yes	t	ca	Rotate ca
tac	Yes	t	ac	Rotate ac
tca	No	t	ca	Rotate tca
cat	No	c	at	Done

Notice that we must rotate back to the starting point with two letters before performing a 3-letter rotation. This leads to sequences like cat, cta, cat. The redundant combinations aren't displayed.

How do we anagram the rightmost n–1 letters? By calling ourselves. The recursive `doAnagram()` method takes the size of the word to be anagrammed as its only parameter. This word is understood to be the rightmost n letters of the complete word. Each time `doAnagram()` calls itself, it does so with a word one letter smaller than before, as shown in Figure 6.7.

The base case occurs when the size of the word to be anagrammed is only one letter. There's no way to rearrange one letter, so the method returns immediately. Otherwise, it anagrams all but the first letter of the word it was given and then rotates the entire word. These two actions are performed n times, where n is the size of the word. Here's the recursive routine `doAnagram()`:

```
public static void doAnagram(int newSize)
   {
   if(newSize == 1)                    // if too small,
      return;                          // go no further
   for(int j=0; j<newSize; j++)        // for each position,
      {
      doAnagram(newSize-1);            // anagram remaining
```

```
      if(newSize==2)                    // if innermost,
         displayWord();                 // display it
      rotate(newSize);                  // rotate word
      }

   }
```

Each time the `doAnagram()` method calls itself, the size of the word is one letter smaller, and the starting position is one cell further to the right, as shown in Figure 6.8.

Figure 6.7: The recursive `doAnagram()` method

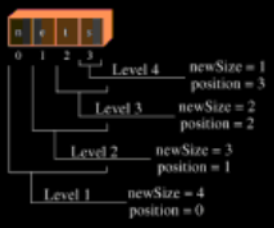

Figure 6.8: Smaller and smaller words

Listing 6.2 shows the complete `anagram.java` program. The `main()` routine gets a word from the user, inserts it into a character array so it can be dealt with conveniently, and then calls `doAnagram()`.

Listing 6.2 The anagram.java Program

```
// anagram.java
// creates anagrams
// to run this program: C>java AnagramApp
import java.io.*;                   // for I/O
////////////////////////////////////////////////////////////////
class AnagramApp
   {
   static int size;
```

```
   static int count;
   static char[] arrChar = new char[100];

   public static void main(String[] args) throws IOException
      {
      System.out.print("Enter a word: ");    // get word
      System.out.flush();
      String input = getString();
      size = input.length();                 // find its size
      count = 0;
      for(int j=0; j<size; j++)              // put it in array
         arrChar[j] = input.charAt(j);
      doAnagram(size);                       // anagram it
      }  // end main()

   //--------------------------------------------------------------
   public static void doAnagram(int newSize)
      {
      if(newSize == 1)                     // if too small,
         return;                           // go no further
      for(int j=0; j<newSize; j++)         // for each
position,
         {
         doAnagram(newSize-1);             // anagram remaining
         if(newSize==2)                    // if innermost,
            displayWord();                 // display it
         rotate(newSize);                  // rotate word
         }
      }

   //--------------------------------------------------------------
   // rotate left all chars from position to end
   public static void rotate(int newSize)
      {
      int j;
      int position = size - newSize;
      char temp = arrChar[position];       // save first letter
      for(j=position+1; j<size; j++)       // shift others left
         arrChar[j-1] = arrChar[j];
      arrChar[j-1] = temp;                 // put first on
right
      }

   //--------------------------------------------------------------
   public static void displayWord()
      {
      if(count < 99)
         System.out.print(" ");
      if(count < 9)
         System.out.print(" ");
      System.out.print(++count + " ");
```

```
      for(int j=0; j<size; j++)
         System.out.print( arrChar[j] );
      System.out.print("   ");
      System.out.flush();
      if(count%6 == 0)
         System.out.println("");
      }

   //-------------------------------------------------------
-
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }

   //-------------------------------------------------------
---

   }  // end class AnagramApp
```

The `rotate()` method rotates the word one position left as described earlier. The `displayWord()` method displays the entire word and adds a count to make it easy to see how many words have been displayed. Here's some sample interaction with the program:

```
Enter a word: cats
1 cats     2 cast     3 ctsa     4 ctas     5 csat     6 csta
7 atsc     8 atcs     9 asct    10 astc    11 acts    12 acst
13 tsca    14 tsac    15 tcas    16 tcsa    17 tasc    18 tacs
19 scat    20 scta    21 satc    22 sact    23 stca    24 stac
```

(Is it only coincidence that scat is an anagram of cats?) You can use the program to anagram 5-letter or even 6-letter words. However, because the factorial of 6 is 720, this may generate more words than you want to know about.

The Towers of Hanoi

The Towers of Hanoi is an ancient puzzle consisting of a number of disks placed on three columns, as shown in Figure 6.10.

The disks all have different diameters and holes in the middle so they will fit over the columns. All the disks start out on column A. The object of the puzzle is to transfer all the disks from column A to column C. Only one disk can be moved at a time, and no disk can be placed on a disk that's smaller than itself.

There's an ancient myth that somewhere in India, in a remote temple, monks labor day and night to transfer 64 golden disks from one of three diamond-studded towers to another. When they are finished, the world will end. Any alarm you may feel, however, will be dispelled when you see how long it takes to solve the puzzle for far fewer than 64 disks.

The Towers Workshop Applet

Start up the Towers Workshop applet. You can attempt to solve the puzzle yourself by using the mouse to drag the topmost disk to another tower. Figure 6.11 shows how this looks after several moves have been made.

There are three ways to use the workshop applet.

- You can attempt to solve the puzzle manually, by dragging the disks from tower to tower.
- You can repeatedly press the Step button to watch the algorithm solve the puzzle. At each step in the solution, a message is displayed, telling you what the algorithm is doing.
- You can press the Run button and watch the algorithm solve the puzzle with no intervention on your part; the disks zip back and forth between the posts.

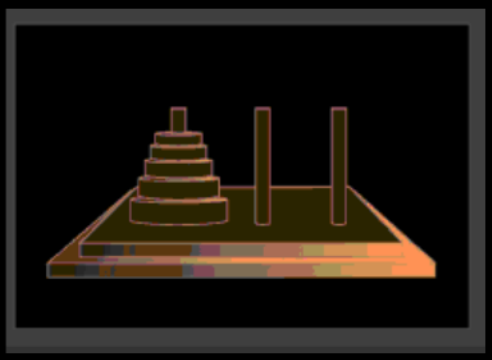

Figure 6.10: The Towers of Hanoi

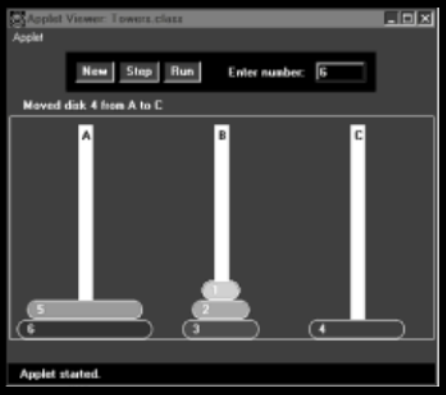

Figure 6.11: The Towers Workshop applet

To restart the puzzle, type in the number of disks you want to use, from 1 to 10, and press New twice. (After the first time, you're asked to verify that restarting is what you want to do.) The specified number of disks will be arranged on tower A. Once you drag a disk with the mouse, you can't use Step or Run; you must start over with New. However, you can switch to manual in the middle of stepping or running, and you can switch to Step when you're running, and Run when you're stepping.

Try solving the puzzle manually with a small number of disks, say 3 or 4. Work up to higher numbers. The applet gives you the opportunity to learn intuitively how the problem is solved.

Moving Subtrees

Let's call the initial tree-shaped (or pyramid-shaped) arrangement of disks on tower A a *tree*. As you experiment with the applet, you'll begin to notice that smaller tree-shaped stacks of disks are generated as part of the solution process. Let's call these smaller trees, containing fewer than the total number of disks, *subtrees*. For example, if you're trying to transfer 4 disks, you'll find that one of the intermediate steps involves a subtree of 3 disks on tower B, as shown in Figure 6.12.

These subtrees form many times in the solution of the puzzle. This is because the creation of a subtree is the only way to transfer a larger disk from one tower to another: all the smaller disks must be placed on an intermediate tower, where they naturally form a subtree.

Figure 6.12: A subtree on tower B

Here's a rule of thumb that may help when you try to solve the puzzle manually. If the subtree you're trying to move has an odd number of disks, start by moving the topmost disk directly to the tower where you want the subtree to go. If you're trying to move a subtree with an even number of disks, start by moving the topmost disk to the intermediate tower.

The Recursive Algorithm

The solution to the Towers of Hanoi puzzle can be expressed recursively using the notion of subtrees. Suppose you want to move all the disks from a source tower (call it S) to a destination tower (call it D). You have an intermediate tower available (call it I). Assume there are n disks on tower S. Here's the algorithm:

1. Move the subtree consisting of the top n–1 disks from S to I.

2. Move the remaining (largest) disk from S to D.

3. Move the subtree from I to D.

When you begin, the source tower is A, the intermediate tower is B, and the destination tower is C. Figure 6.13 shows the three steps for this situation.

First, the subtree consisting of disks 1, 2, and 3 is moved to the intermediate tower B. Then the largest disk, 4, is moved to tower C. Then the subtree is moved from B to C.

Of course, this doesn't solve the problem of how to move the subtree consisting of disks 1, 2, and 3 to tower B, because you can't move a subtree all at once; you must move it one disk at a time. Moving the 3-disk subtree is not so easy. However, it's easier than moving 4 disks.

As it turns out, moving 3 disks from A to the destination tower B can be done with the same 3 steps as moving 4 disks. That is, move the subtree consisting of the top 2 disks from tower A to intermediate tower C; then move disk 3 from A to B. Then move the subtree back from C to B.

How do you move a subtree of two disks from A to C? Move the subtree consisting of only one disk (1) from A to B. This is the base case: when you're moving only one disk, you just move it; there's nothing else to do. Then move the larger disk (2) from A to C, and replace the subtree (disk 1) on it.

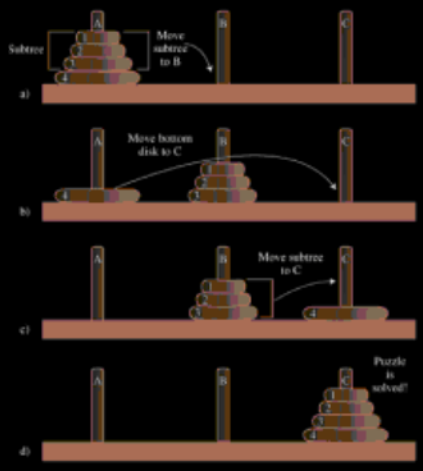

Figure 6.13: Recursive solution to Towers puzzle

The `towers.java` Program

The `towers.java` program solves the Towers of Hanoi puzzle using this recursive approach. It communicates the moves by displaying them; this requires much less code than displaying the towers. It's up to the human reading the list to actually carry out the moves.

The code is simplicity itself. The `main()` routine makes a single call to the recursive method `doTowers()`. This method then calls itself recursively until the puzzle is solved. In this version, shown in Listing 6.4, there are initially only 3 disks, but you can recompile the program with any number.

Listing 6.4 The `towers.java` Program

```
// towers.java
// evaluates triangular numbers
// to run this program: C>java TowersApp
import java.io.*;                  // for I/O
////////////////////////////////////////////////////////////////
class TowersApp
   {
   static int nDisks = 3;

   public static void main(String[] args)
      {
      doTowers(nDisks, 'A', 'B', 'C');
      }

   //-------------------------------------------------------------
-
   public static void doTowers(int topN,
                               char from, char inter, char to)
```

```
        {
        if(topN==1)
           System.out.println("Disk 1 from " + from + " to "+
to);
        else
           {
           doTowers(topN-1, from, to, inter);  // from-->inter

           System.out.println("Disk " + topN +
                              " from " + from + " to "+ to);
           doTowers(topN-1, inter, from, to);  // inter-->to
           }
        }

     //--------------------------------------------------------

     }  // end class TowersApp
```

Remember that 3 disks are moved from A to C. Here's the output from the program:

```
Disk 1 from A to C
Disk 2 from A to B
Disk 1 from C to B
Disk 3 from A to C
Disk 1 from B to A
Disk 2 from B to C
Disk 1 from A to C
```

The arguments to `doTowers()` are the number of disks to be moved, and the source (`from`), intermediate (`inter`), and destination (`to`) towers to be used. The number of disks decreases by 1 each time the method calls itself. The source, intermediate, and destination towers also change.

Here is the output with additional notations that show when the method is entered and when it returns, its arguments, and whether a disk is moved because it's the base case (a subtree consisting of only one disk) or because it's the remaining bottom disk after a subtree has been moved.

```
Enter (3 disks): s=A, i=B, d=C
   Enter (2 disks): s=A, i=C, d=B
      Enter (1 disk): s=A, i=B, d=C
         Base case: move disk 1 from A to C
      Return (1 disk)
      Move bottom disk 2 from A to B

      Enter (1 disk): s=C, i=A, d=B
         Base case: move disk 1 from C to B
      Return (1 disk)
   Return (2 disks)
   Move bottom disk 3 from A to C
   Enter (2 disks): s=B, i=A, d=C
      Enter (1 disk): s=B, i=C, d=A
         Base case: move disk 1 from B to A
      Return (1 disk)
```

```
         Move bottom disk 2 from B to C
         Enter (1 disk): s=A, i=B, d=C
            Base case: move disk 1 from A to C
         Return (1 disk)
      Return (2 disks)
   Return (3 disks)
```

If you study this output along with the source code for `doTower()`, it should become clear exactly how the method works. It's amazing that such a small amount of code can solve such a seemingly complicated problem.

Mergesort

Our final example of recursion is the mergesort. This is a much more efficient sorting technique than those we saw in Chapter 3, "Simple Sorting," at least in terms of speed. While the bubble, insertion, and selection sorts take $O(N^2)$ time, the mergesort is O(N*logN). The graph in Figure 2.9 (in Chapter 2) shows how much faster this is. For example, if N (the number of items to be sorted) is 10,000, then N^2 is 100,000,000, while N*logN is only 40,000. If sorting this many items required 40 seconds with the mergesort, it would take almost 28 hours for the insertion sort.

The mergesort is also fairly easy to implement. It's conceptually easier than quicksort and the Shell short, which we'll encounter in the next chapter.

The downside of the mergesort is that it requires an additional array in memory, equal in size to the one being sorted. If your original array barely fits in memory, the mergesort won't work. However, if you have enough space, it's a good choice.

Merging Two Sorted Arrays

The heart of the mergesort algorithm is the merging of two already sorted arrays. Merging two sorted arrays A and B creates a third array, C, that contains all the elements of A and B, also arranged in sorted order. We'll examine the merging process first; later we'll see how it's used in sorting.

Imagine two sorted arrays. They don't need to be the same size. Let's say array A has 4 elements and array B has 6. They will be merged into an array C that starts with 10 empty cells. Figure 6.14 shows how this looks.

In the figure, the circled numbers indicate the order in which elements are transferred from A and B to C. Table 6.3 shows the comparisons necessary to determine which element will be copied. The steps in the table correspond to the steps in the figure. Following each comparison, the smaller element is copied to A.

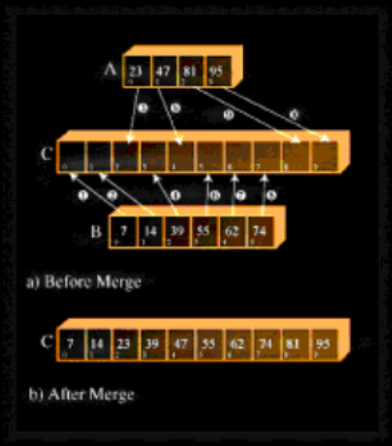

Figure 6.14: Merging two arrays

Table 6.3: Merging Operations

Step	Comparison (If Any)	Copy
1	Compare 23 and 7	Copy 7 from B to C
2	Compare 23 and 14	Copy 14 from B to C
3	Compare 23 and 39	Copy 23 from A to C
4	Compare 39 and 47	Copy 39 from B to C
5	Compare 55 and 47	Copy 47 from A to C
6	Compare 55 and 81	Copy 55 from B to C
7	Compare 62 and 81	Copy 62 from B to C
8	Compare 74 and 81	Copy 74 from B to C
9		Copy 81 from A to C
10		Copy 95 from A to C

Notice that, because B is empty following step 8, no more comparisons are necessary; all the remaining elements are simply copied from A into C.

Listing 6.5 shows a Java program that carries out the merge shown in Figure 6.14 and Table 6.3.

Listing 6.5 The merge.java Program

```
// merge.java
// demonstrates merging two arrays into a third
// to run this program: C>java MergeApp
////////////////////////////////////////////////////////////////
class MergeApp
   {
   public static void main(String[] args)
      {
      int[] arrayA = {23, 47, 81, 95};
      int[] arrayB = {7, 14, 39, 55, 62, 74};
      int[] arrayC = new int[10];

      merge(arrayA, 4, arrayB, 6, arrayC);
      display(arrayC, 10);
      }  // end main()

   //-------------------------------------------------------------
-
                                       // merge A and B into C
     public static void merge( int[] arrayA, int sizeA,
                               int[] arrayB, int sizeB,
                               int[] arrayC )
      {
      int aDex=0, bDex=0, cDex=0;

      while(aDex < sizeA && bDex < sizeB)  // neither array
empty
         if( arrayA[aDex] < arrayB[bDex] )
            arrayC[cDex++] = arrayA[aDex++];
         else
            arrayC[cDex++] = arrayB[bDex++];

      while(aDex < sizeA)                  // arrayB is empty,
         arrayC[cDex++] = arrayA[aDex++];  // but arrayA isn't

      while(bDex < sizeB)                  // arrayA is empty,
         arrayC[cDex++] = arrayB[bDex++];  // but arrayB isn't
      }  // end merge()

   //-------------------------------------------------------------
-
                                       // display array
   public static void display(int[] theArray, int size)
      {
      for(int j=0; j<size; j++)
         System.out.print(theArray[j] + " ");
      System.out.println("");
      }

   //-------------------------------------------------------------
-
```

```
    }  // end class MergeApp
```

In `main()` the arrays `arrayA`, `arrayB`, and `arrayC` are created; then the `merge()` method is called to merge `arrayA` and `arrayB` into `arrayC`, and the resulting contents of `arrayC` are displayed. Here's the output:

```
   7 14 23 39 47 55 62 74 81 95
```

The `merge()` method has three `while` loops. The first steps along both `arrayA` and `arrayB`, comparing elements and copying the smaller of the two into `arrayC`.

The second `while` loop deals with the situation when all the elements have been transferred out of `arrayB`, but `arrayA` still has remaining elements. (This is what happens in the example, where 81 and 95 remain in `arrayA`.) The loop simply copies the remaining elements from `arrayA` into `arrayC`.

The third loop handles the similar situation when all the elements have been transferred out of `arrayA` but `arrayB` still has remaining elements; they are copied to `arrayC`.

Sorting by Merging

The idea in the mergesort is to divide an array in half, sort each half, and then use the `merge()` method to merge the two halves into a single sorted array. How do you sort each half? This chapter is about recursion, so you probably already know the answer: You divide the half into two quarters, sort each of the quarters, and merge them to make a sorted half.

Similarly, each pair of 8ths is merged to make a sorted quarter, each pair of 16ths is merged to make a sorted 8th, and so on. You divide the array again and again until you reach a subarray with only one element. This is the base case; it's assumed an array with one element is already sorted.

We've seen that generally something is reduced in size each time a recursive method calls itself, and built back up again each time the method returns. In `mergeSort()` the range is divided in half each time this method calls itself, and each time it returns it merges two smaller ranges into a larger one.

As `mergeSort()` returns from finding 2 arrays of 1 element each, it merges them into a sorted array of 2 elements. Each pair of resulting 2-element arrays is then merged into a 4-element array. This process continues with larger and larger arrays until the entire array is sorted. This is easiest to see when the original array size is a power of 2, as shown in Figure 6.15.

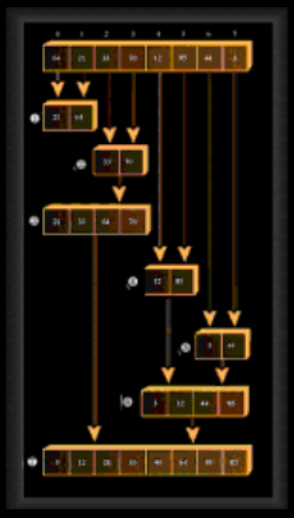

Figure 6.15: Merging larger and larger arrays

First, in the bottom half of the array, range 0-0 and range 1-1 are merged into range 0-1. Of course, 0-0 and 1-1 aren't really ranges; they're only one element, so they are base cases. Similarly, 2-2 and 3-3 are merged into 2-3. Then ranges 0-1 and 2-3 are merged 0-3.

In the top half of the array, 4-4 and 5-5 are merged into 4-5, 6-6 and 7-7 are merged into 6-7, and 4-5 and 6-7 are merged into 4-7. Finally the top half, 0-3, and the bottom half, 4-7, are merged into the complete array, 0-7, which is now sorted.

When the array size is not a power of 2, arrays of different sizes must be merged. For example, Figure 6.16 shows the situation in which the array size is 12. Here an array of size 2 must be merged with an array of size 1 to form an array of size 3.

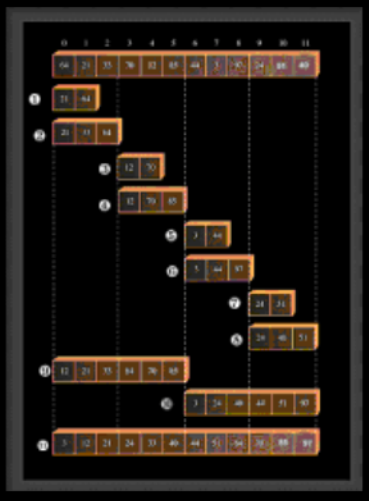

Figure 6.16: Array size not a power of 2

First the 1-element ranges 0-0 and 1-1 are merged into the 2-element range 0-1. Then range 0-1 is merged with the 1-element range 2-2. This creates a 3-element range 0-2. It's merged with the 3-element range 3-5. The process continues until the array is sorted.

Notice that in mergesort we don't merge two separate arrays into a third one, as we

demonstrated in the `merge.java` program. Instead, we merge parts of a single array into itself.

You may wonder where all these subarrays are located in memory. In the algorithm, a workspace array of the same size as the original array is created. The subarrays are stored in sections of the workspace array. This means that subarrays in the original array are copied to appropriate places in the workspace array. After each merge, the workspace array is copied back into the original array.

The MERGESORT Workshop Applet

All this is easier to appreciate when you see it happening before your very eyes. Start up the mergeSort Workshop applet. Repeatedly pressing the Step button will execute mergeSort step by step. Figure 6.17 shows what it looks like after the first three presses.

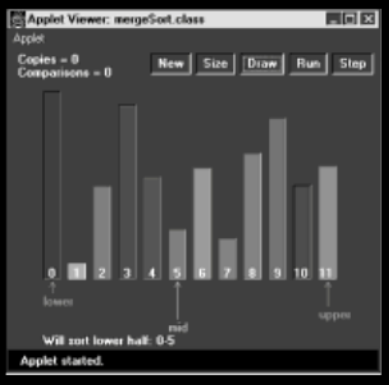

Figure 6.17: The mergeSort Workshop applet

The Lower and Upper arrows show the range currently being considered by the algorithm, and the Mid arrow shows the middle part of the range. The range starts as the entire array and then is halved each time the `mergeSort()` method calls itself. When the range is one element, `mergeSort()` returns immediately; that's the base case. Otherwise, the two subarrays are merged. The applet provides messages, such as `Entering mergeSort: 0-5`, to tell you what it's doing and the range it's operating on.

Many steps involve the `mergeSort()` method calling itself or returning. Comparisons and copies are performed only during the merge process, when you'll see messages such as `Merged 0-0 and 1-1 into workspace`. You can't see the merge happening, because the workspace isn't shown. However, you can see the result when the appropriate section of the workspace is copied back into the original (visible) array: The bars in the specified range will appear in sorted order.

First, the first 2 bars will be sorted, then the first 3 bars, then the 2 bars in the range 3-4, then the 3 bars in the range 3-5, then the 6 bars in the range 0-5, and so on, corresponding to the sequence shown in Figure 6.16. Eventually all the bars will be sorted.

You can cause the algorithm to run continuously by pressing the Run button. You can stop this process at any time by pressing Step, single-step as many times as you want, and resume running by pressing Run again.

As in the other sorting Workshop applets, pressing New resets the array with a new group of unsorted bars and toggles between random and inverse arrangements. The Size button toggles between 12 bars and 100 bars.

It's especially instructive to watch the algorithm run with 100 inversely sorted bars. The

resulting patterns show clearly how each range is sorted individually and merged with its other half, and how the ranges grow larger and larger.

The `mergeSort.java` Program

In a moment we'll look at the entire `mergeSort.java` program. First, let's focus on the method that carries out the mergesort. Here it is:

```
   private void recMergeSort(double[] workSpace, int lowerBound,
                                                 int upperBound)
      {
      if(lowerBound == upperBound)            // if range is 1,
         return;                              // no use sorting
      else
         {                                    // find midpoint
         int mid = (lowerBound+upperBound) / 2;
                                              // sort low half
         recMergeSort(workSpace, lowerBound, mid);
                                              // sort high half
         recMergeSort(workSpace, mid+1, upperBound);
                                              // merge them
         merge(workSpace, lowerBound, mid+1, upperBound);
         }  // end else

      }  // end recMergeSort
```

As you can see, beside the base case, there are only four statements in this method. One computes the midpoint, there are two recursive calls to `recMergeSort()` (one for each half of the array), and finally a call to `merge()` to merge the two sorted halves. The base case occurs when the range contains only one element (`lowerBound==upperBound`) and results in an immediate return.

In the `mergeSort.java` program, the `mergeSort()` method is the one actually seen by the class user. It creates the array `workSpace[]`, and then calls the recursive routine `recMergeSort()` to carry out the sort. The creation of the workspace array is handled in `mergeSort()` because doing it in `recMergeSort()` would cause the array to be created anew with each recursive call, an inefficiency.

The `merge()` method in the previous `merge.java` program operated on three separate arrays: two source arrays and a destination array. The `merge()` routine in the `mergeSort.java` program operates on a single array: the `theArray` member of the `DArray` class. The arguments to this `merge()` method are the starting point of the low-half subarray, the starting point of the high-half subarray, and the upper bound of the high-half subarray. The method calculates the sizes of the subarrays based on this information.

Listing 6.6 shows the complete `mergeSort.java` program. This program uses a variant of the array classes from Chapter 2, adding the `mergeSort()` and `recMergeSort()` methods to the `DArray` class. The `main()` routine creates an array, inserts 12 items, displays the array, sorts the items with `mergeSort()`, and displays the array again.

Listing 6.6 The mergeSort.java Program

```
// mergeSort.java
// demonstrates recursive mergesort
```

```
// to run this program: C>java MergeSortApp
import java.io.*;                   // for I/O
////////////////////////////////////////////////////////////////
class DArray
   {
   private double[] theArray;       // ref to array theArray
   private int nElems;              // number of data items

   //--------------------------------------------------------
-
   public DArray(int max)           // constructor
      {
      theArray = new double[max];   // create array
      nElems = 0;
      }

   //--------------------------------------------------------
-
   public void insert(double value)  // put element into array
      {
      theArray[nElems] = value;     // insert it
      nElems++;                     // increment size
      }

   //--------------------------------------------------------
-
   public void display()            // displays array contents
      {
      for(int j=0; j<nElems; j++)    // for each element,
         System.out.print(theArray[j] + " ");  // display it
      System.out.println("");
      }

   //--------------------------------------------------------
-
   public void mergeSort()          // called by main()
      {                             // provides workspace
      double[] workSpace = new double[nElems];
      recMergeSort(workSpace, 0, nElems-1);
      }

   //--------------------------------------------------------
-
   private void recMergeSort(double[] workSpace, int
lowerBound,
                                          int upperBound)
      {
      if(lowerBound == upperBound)           // if range is 1,
         return;                             // no use sorting
      else
         {                                   // find midpoint
         int mid = (lowerBound+upperBound) / 2;
                                             // sort low half
         recMergeSort(workSpace, lowerBound, mid);
```

```
                                          // sort high half
         recMergeSort(workSpace, mid+1, upperBound);
                                          // merge them
         merge(workSpace, lowerBound, mid+1, upperBound);
         }  // end else
      }  // end recMergeSort

   //--------------------------------------------------------
-
   private void merge(double[] workSpace, int lowPtr,
                    int highPtr, int upperBound)
      {
      int j = 0;                          // workspace index
      int lowerBound = lowPtr;
      int mid = highPtr-1;
      int n = upperBound-lowerBound+1;       // # of items

      while(lowPtr <= mid && highPtr <= upperBound)
         if( theArray[lowPtr] < theArray[highPtr] )
            workSpace[j++] = theArray[lowPtr++];
         else
            workSpace[j++] = theArray[highPtr++];

      while(lowPtr <= mid)
         workSpace[j++] = theArray[lowPtr++];

      while(highPtr <= upperBound)
         workSpace[j++] = theArray[highPtr++];

      for(j=0; j<n; j++)
         theArray[lowerBound+j] = workSpace[j];
      }  // end merge()

   //--------------------------------------------------------
-
   }  // end class DArray

////////////////////////////////////////////////////////////////

class MergeSortApp
   {
   public static void main(String[] args)
      {
      int maxSize = 100;             // array size
      DArray arr;                    // reference to array
      arr = new DArray(maxSize);     // create the array

      arr.insert(64);                // insert items
      arr.insert(21);
      arr.insert(33);
      arr.insert(70);
      arr.insert(12);
      arr.insert(85);
```

```
        arr.insert(44);
        arr.insert(3);
        arr.insert(99);
        arr.insert(0);
        arr.insert(108);
        arr.insert(36);

        arr.display();                  // display items

        arr.mergeSort();                // mergesort the array

        arr.display();                  // display items again
        }  // end main()

    }  // end class MergeSortApp
```

The output from the program is simply the display of the unsorted and sorted arrays:

```
64 21 33 70 12 85 44 3 99 0 108 36
0 3 12 21 33 36 44 64 70 85 99 108
```

If we put additional statements in the `recMergeSort()` method, we could generate a running commentary on what the program does during a sort. The following output shows how this might look for the 4-item array {64, 21, 33, 70}. (You can think of this as the lower half of the array in Figure 6.15.)

```
Entering 0-3
   Will sort low half of 0-3
   Entering 0-1
      Will sort low half of 0-1
         Entering 0-0
         Base-Case Return 0-0
      Will sort high half of 0-1
         Entering 1-1
         Base-Case Return 1-1
      Will merge halves into 0-1
   Return 0-1                       theArray=21 64 33 70
   Will sort high half of 0-3
   Entering 2-3
      Will sort low half of 2-3
         Entering 2-2
         Base-Case Return 2-2
      Will sort high half of 2-3
         Entering 3-3
         Base-Case Return 3-3
      Will merge halves into 2-3
   Return 2-3                       theArray=21 64 33 70
   Will merge halves into 0-3

Return 0-3                          theArray=21 33 64 70
```

This is roughly the same content as would be generated by the mergeSort Workshop applet if it could sort 4 items. Study of this output, and comparison with the code for

`recMergeSort()` and Figure 6.15, will reveal the details of the sorting process.

Efficiency of the Mergesort

As we noted, the mergesort runs in O(N*logN) time. How do we know this? Let's see how we can figure out the number of times a data item must be copied, and the number of times it must be compared with another data item, during the course of the algorithm. We assume that copying and comparing are the most time-consuming operations; that the recursive calls and returns don't add much overhead.

Number of Copies

Consider Figure 6.15. Each cell below the top line represents an element copied from the array into the workspace.

Adding up all the cells in Figure 6.15 (the 7 numbered steps) shows there are 24 copies necessary to sort 8 items. $Log_2 8$ is 3, so $8*log_2 8$ equals 24. This shows that, for the case of 8 items, the number of copies is proportional to $N*log_2 N$.

Another way to look at this is that, to sort 8 items requires 3 *levels*, each of which involves 8 copies. A level means all copies into the same size subarray. In the first level, there are 4 2-element subarrays; in the second level, there are 2 4-element subarrays; and in the third level, there is 1 8-element subarray. Each level has 8 elements, so again there are 3*8 or 24 copies.

In Figure 6.15, by considering only half the graph, you can see that 8 copies are necessary for an array of 4 items (steps 1, 2, and 3), and 2 copies are necessary for 2 items. Similar calculations provide the number of copies necessary for larger arrays. Table 6.4 summarizes this information.

Table 6.4: Number of Operations When N is a Power of 2

N	log2N	Number of Copies into Workspace (N*log2N)	Total Copies	Comparisons Max (Min)
2	1	2	4	1 (1)
4	2	8	16	5 (4)
8	3	24	48	17 (12)
16	4	64	128	49 (32)
32	5	160	320	129 (80)
64	6	384	768	321 (192)
128	7	896	1792	769 (448)

Actually, the items are not only copied into the workspace, they're also copied back into

the original array. This doubles the number of copies, as shown in the Total Copies column. The final column of Table 6.4 shows comparisons, which we'll return to in a moment.

It's harder to calculate the number of copies and comparisons when N is not a multiple of 2, but these numbers fall between those that are a power of 2. For 12 items, there are 88 total copies, and for 100 items, 1344 total copies.

Number of Comparisons

In the mergesort algorithm, the number of comparisons is always somewhat less than the number of copies. How much less? Assuming the number of items is a power of 2, for each individual merging operation, the maximum number of comparisons is always one less than the number of items being merged, and the minimum is half the number of items being merged. You can see why this is true in Figure 6.18, which shows two possibilities when trying to merge 2 arrays of 4 items each.

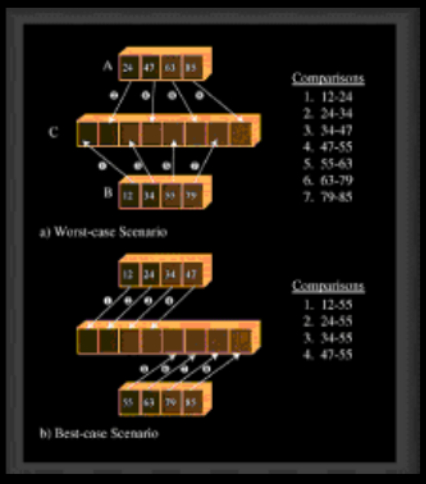

Figure 6.18: Maximum and minimum comparisons

In the first case, the items interleave, and 7 comparisons must be made to merge them. In the second case, all the items in one array are smaller than all the items in the other, so only 4 comparisons must be made.

There are many merges for each sort, so we must add the comparisons for each one. Referring to Figure 6.15, you can see that 7 merge operations are required to sort 8 items. The number of items being merged and the resulting number of comparisons is shown in Table 6.5.

Table 6.5: Comparisons Involved in Sorting 8 Items

Step Number	1	2	3	4	5	6	7	Totals
Number of items being merged(N)	2	2	4	2	2	4	8	24
Maximum comparisons(N–1)	1	1	3	1	1	3	7	17
Minimum	1	1	2	1	1	2	4	12

comparisons(N/2)

For each merge, the maximum number of comparisons is one less than the number of items. Adding these figures for all the merges gives us a total of 17.

The minimum number of comparisons is always half the number of items being merged, and adding these figures for all the merges results in 12 comparisons. Similar arithmetic results in the Comparisons columns for Table 6.4. The actual number of comparisons to sort a specific array depends on how the data is arranged; but it will be somewhere between the maximum and minimum values.

Eliminating Recursion

Some algorithms lend themselves to a recursive approach, some don't. As we've seen, the recursive `triangle()` and `factorial()` methods can be implemented more efficiently using a simple loop. However, various divide-and-conquer algorithms, such as mergesort, work very well as a recursive routine.

Often an algorithm is easy to conceptualize as a recursive method, but in practice the recursive approach proves to be inefficient. In such cases, it's useful to transform the recursive approach into a nonrecursive approach. Such a transformation can often make use of a stack.

Recursion and Stacks

There is a close relationship between recursion and stacks. In fact, most compilers implement recursion by using stacks. As we noted, when a method is called, they push the arguments to the method and the return address (where control will go when the method returns) on the stack, and then transfer control to the method. When the method returns, they pop these values off the stack. The arguments disappear, and control returns to the return address.

Simulating a Recursive Method

In this section we'll demonstrate how any recursive solution can be transformed into a stack-based solution. Remember the recursive `triangle()` method from the first section in this chapter? Here it is again:

```
int triangle(int n)
   {
   if(n==1)
      return 1;
   else
      return( n + triangle(n-1) );

   }
```

We're going to break this algorithm down into its individual operations, making each operation one `case` in a `switch` statement. (You can perform a similar decomposition using `goto` statements in C++ and some other languages, but Java doesn't support `goto`.)

The `switch` statement is enclosed in a method called `step()`. Each call to `step()` causes one `case` section within the `switch` to be executed. Calling `step()` repeatedly will eventually execute all the code in the algorithm.

The `triangle()` method we just saw performs two kinds of operations. First, it carries out the arithmetic necessary to compute triangular numbers. This involves checking if `n` is 1, and adding `n` to the results of previous recursive calls. However, `triangle()` also performs the operations necessary to manage the method itself. These involve transfer of control, argument access, and the return address. These operations are not visible by looking at the code; they're built into all methods. Here, roughly speaking, is what happens during a call to a method:

- When a method is called, its arguments and the return address are pushed onto a stack.
- A method can access its arguments by peeking at the top of the stack.
- When a method is about to return, it peeks at the stack to obtain the return address, and then pops both this address and its arguments off the stack and discards them.

The `stackTriangle.java` program contains three classes: `Params`, `StackX`, and `StackTriangleApp`. The `Params` class encapsulates the return address and the method's argument, `n`; objects of this class are pushed onto the stack. The `StackX` class is similar to those in other chapters, except that it holds objects of class `Params`. The `StackTriangleApp` class contains four methods: `main()`, `recTriangle()`, `step()`, and the usual `getInt()` method for numerical input.

The `main()` routine asks the user for a number, calls the `recTriangle()` method to calculate the triangular number corresponding to `n`, and displays the result.

The `recTriangle()` method creates a `StackX` object and initializes `codePart` to 1. It then settles into a `while` loop where it repeatedly calls `step()`. It won't exit from the loop until `step()` returns `true` by reaching case 6, its exit point. The `step()` method is basically a large `switch` statement in which each `case` corresponds to a section of code in the original `triangle()` method. Listing 6.7 shows the `stackTriangle.java` program.

Listing 6.7 The stackTriangle.java Program

```
// stackTriangle.java
// evaluates triangular numbers, stack replaces recursion
// to run this program: C>java StackTriangleApp
import java.io.*;                // for I/O
////////////////////////////////////////////////////////////////
class Params     // parameters to save on stack
   {
   public int n;
   public int codePart;

   public Params(int nn, int ra)
      {
      n=nn;
      returnAddress = ra;
      }
   }  // end class Params

////////////////////////////////////////////////////////////////
```

```
class StackX
   {
   private int maxSize;        // size of stack array
   private Params[] stackArray;
   private int top;            // top of stack

//--------------------------------------------------------------
-
   public StackX(int s)         // constructor
      {
      maxSize = s;             // set array size
      stackArray = new Params[maxSize];  // create array
      top = -1;                // no items yet
      }

//--------------------------------------------------------------
-
   public void push(Params p)  // put item on top of stack
      {
      stackArray[++top] = p;     // increment top, insert item
      }

//--------------------------------------------------------------
-
   public Params pop()         // take item from top of stack
      {
      return stackArray[top--];  // access item, decrement top
      }

//--------------------------------------------------------------
-
   public Params peek()        // peek at top of stack
      {
      return stackArray[top];
      }

//--------------------------------------------------------------
-
   }  // end class StackX

////////////////////////////////////////////////////////////////

class StackTriangleApp
   {
   static int theNumber;
   static int theAnswer;
   static StackX theStack;
   static int codePart;
   static Params theseParams;

   public static void main(String[] args) throws IOException
      {
      System.out.print("Enter a number: ");
      System.out.flush();
```

```
      theNumber = getInt();
      triangle();
      System.out.println("Triangle="+theAnswer);
      }  // end main()

//--------------------------------------------------------------
   public static void recTriangle()
      {
      theStack = new StackX(50);
      codePart = 1;
      while( step() == false)  // call step() until it's true
         ;                     // null statement
      }

//--------------------------------------------------------------
   public static boolean step()
      {
      switch(codePart)
         {
         case 1:                              // initial call
            theseParams = new Params(theNumber, 6);
            theStack.push(theseParams);
            codePart = 2;
            break;
         case 2:                              // method entry
            theseParams = theStack.peek();
            if(theseParams.n == 1)            // test
               {
               theAnswer = 1;
               codePart = 5;   // exit
               }
            else
               codePart = 3;   // recursive call
            break;
         case 3:                              // method call
            Params newParams = new Params(theseParams.n - 1,
4);
            theStack.push(newParams);
            codePart = 2;  // go enter method
            break;
         case 4:                              // calculation
            theseParams = theStack.peek();
            theAnswer = theAnswer + theseParams.n;
            codePart = 5;
            break;
         case 5:                              // method exit
            theseParams = theStack.peek();
            codePart = theseParams.returnAddress; // (4 or 6)
            theStack.pop();
            break;
         case 6:                              // return point
            return true;
         }  // end switch
```

```
      return false;                          // all but 7
      }  // end triangle

//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }

//--------------------------------------------------------------
-
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }

//--------------------------------------------------------------
-

   }  // end class StackTriangleApp
```

This program calculates triangular numbers, just as the `triangle.java` program at the beginning of the chapter did. Here's some sample output:

```
Enter a number: 100
Triangle=5050
```

Figure 6.19 shows how the sections of code in each `case` relate to the various parts of the algorithm.

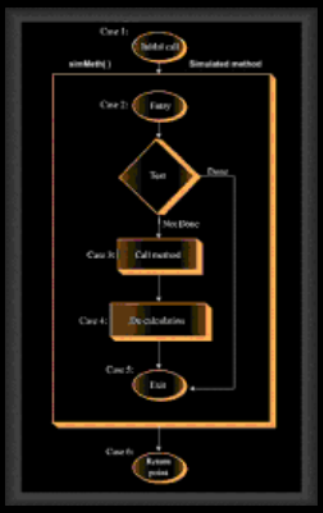

Figure 6.19: The `cases` and the `step()` method

The program simulates a method, but it has no name in the listing because it isn't a real

Java method. Let's call this simulated method `simMeth()`. The initial call to `simMeth()` (at case 1) pushes the value entered by the user and a return value of 6 onto the stack and moves to the entry point of `simMeth()` (case 2).

At its entry (case 2), `simMeth()` tests whether its argument is 1. It accesses the argument by peeking at the top of the stack. If the argument is 1, this is the base case and control goes to `simMeth()`'s exit (case 5). If not, it calls itself recursively (case 3). This recursive call consists of pushing `n-1` and a return address of 4 onto the stack, and going to the method entry at case 2.

On the return from the recursive call, `simMeth()` adds its argument `n` to the value returned from the call. Finally it exits (case 5). When it exits, it pops the last `Params` object off the stack; this information is no longer needed.

The return address given in the initial call was 6, so case 6 is where control goes when the method returns. This code returns `true` to let the `while` loop in `recTriangle()` know that the loop is over.

Note that in this description of `simMeth()`'s operation we use terms like *argument*, *recursive call*, and *return address* to mean simulations of these features, not the normal Java versions.

If you inserted some output statements in each `case` to see what `simMeth()` was doing, you could arrange for output like this:

```
Enter a number: 4
case 1. theAnswer=0  Stack:
case 2. theAnswer=0  Stack: (4, 6)
case 3. theAnswer=0  Stack: (4, 6)
case 2. theAnswer=0  Stack: (4, 6) (3, 4)
case 3. theAnswer=0  Stack: (4, 6) (3, 4)
case 2. theAnswer=0  Stack: (4, 6) (3, 4) (2, 4)
case 3. theAnswer=0  Stack: (4, 6) (3, 4) (2, 4)
case 2. theAnswer=0  Stack: (4, 6) (3, 4) (2, 4) (1, 4)
case 5. theAnswer=1  Stack: (4, 6) (3, 4) (2, 4) (1, 4)
case 4. theAnswer=1  Stack: (4, 6) (3, 4) (2, 4)
case 5. theAnswer=3  Stack: (4, 6) (3, 4) (2, 4)
case 4. theAnswer=3  Stack: (4, 6) (3, 4)
case 5. theAnswer=6  Stack: (4, 6) (3, 4)
case 4. theAnswer=6  Stack: (4, 6)
case 5. theAnswer=10 Stack: (4, 6)
case 6. theAnswer=10 Stack:
Triangle=10
```

The case number shows what section of code is being executed. The contents of the stack (consisting of `Params` objects containing `n` followed by a return address) are also shown. The `simMeth()` method is entered 4 times (case 2) and returns 4 times (case 5). It's only when it starts returning that `theAnswer` begins to accumulate the results of the calculations.

What Does This Prove?

In `stackTriangle.java` we have a program that more or less systematically transforms a program that uses recursion into a program that uses a stack. This suggests that such a transformation is possible for any program that uses recursion, and in fact this is the case.

With some additional work, you can systematically refine the code we show here, simplifying it and even eliminating the `switch` statement entirely to make the code more efficient.

In practice, however, it's usually more practical to rethink the algorithm from the beginning, using a stack-based approach instead of a recursive approach. Listing 6.8 shows what happens when we do that with the `triangle()` method.

Listing 6.8 The stackTriangle2.java Program

```
// stackTriangle2.java
// evaluates triangular numbers, stack replaces recursion
// to run this program: C>java StackTriangle2App
import java.io.*;                 // for I/O
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;        // size of stack array
   private int[] stackArray;
   private int top;            // top of stack

//--------------------------------------------------------------
-
   public StackX(int s)         // constructor
      {
      maxSize = s;
      stackArray = new int[maxSize];
      top = -1;
      }

//--------------------------------------------------------------
-
   public void push(int p)     // put item on top of stack
      { stackArray[++top] = p; }

//--------------------------------------------------------------
-
   public int pop()            // take item from top of stack
      { return stackArray[top--]; }

//--------------------------------------------------------------
-
   public int peek()           // peek at top of stack
      { return stackArray[top]; }

//--------------------------------------------------------------
-
   public boolean isEmpty()    // true if stack is empty
      { return (top == -1); }

//--------------------------------------------------------------
-
   }  // end class StackX
```

```
////////////////////////////////////////////////////////////////

class StackTriangle2App
   {
   static int theNumber;
   static int theAnswer;
   static StackX theStack;

   public static void main(String[] args) throws IOException
      {
      System.out.print("Enter a number: ");
      System.out.flush();
      theNumber = getInt();
      stackTriangle();
      System.out.println("Triangle="+theAnswer);
      }  // end main()

//--------------------------------------------------------------
   public static void stackTriangle()
      {
      theStack = new StackX(10000);     // make a stack

      theAnswer = 0;                    // initialize answer

      while(theNumber > 0)              // until n is 1,
         {
         theStack.push(theNumber);      // push value
         --theNumber;                   // decrement value
         }
      while( !theStack.isEmpty() )      // until stack empty,
         {
         int newN = theStack.pop();     // pop value,
         theAnswer += newN;             // add to answer
         }
      }

//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }

//--------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
```

```
//--------------------------------------------------------------

   }  // end class StackTriangle2App
```

Here two short `while` loops in the `stackTriangle()` method substitute for the entire `step()` method of the `stackTriangle.java` program. Of course, in this program you can see by inspection that you can eliminate the stack entirely and use a simple loop. However, in more complicated algorithms the stack must remain.

Often you'll need to experiment to see whether a recursive method, a stack-based approach, or a simple loop is the most efficient (or practical) way to handle a particular situation.

Part III

Chapter List

Chapter 7: Advanced Sorting

Overview

We discussed simple sorting in Chapter 3. The sorts described there—the bubble, selection, and insertion sorts—are easy to implement but are rather slow. In Chapter 6 we described the mergesort. It runs much faster than the simple sorts, but requires twice as much space as the original array; this is often a serious drawback.

This chapter covers two advanced approaches to sorting: Shellsort and quicksort. These sorts both operate much faster than the simple sorts; the Shellsort in about $O(N*(\log N)^2)$ time, and quicksort in $O(N*\log N)$ time, which is the fastest time for general-purpose sorts. Neither of these sorts requires a large amount of extra space, as mergesort does. The Shellsort is almost as easy to implement as mergesort, while quicksort is the fastest of all the general-purpose sorts.

We'll examine the Shellsort first. Quicksort is based on the idea of partitioning, so we'll then examine partitioning separately, before examining quicksort itself.

Shellsort

The Shellsort is named for Donald L. Shell, the computer scientist who discovered it in 1959. It's based on the insertion sort but adds a new feature that dramatically improves the insertion sort's performance.

The Shellsort is good for medium-sized arrays, perhaps up to a few thousand items, depending on the particular implementation. (However, see the cautionary notes in Chapter 15 about how much data can be handled by a particular algorithm.) It's not quite